第 2 章　第一个入门范例：helloapp 应用

本章讲解了一个简单的运用 Spring MVC 框架的范例：helloapp 应用，这个例子可以帮助读者迅速入门，获得开发 Spring MVC 的基本经验。该应用的功能非常简单，接收用户输入的姓名<userName>，然后输出“Hello <userName>”。开发 helloapp 应用涉及以下内容：

- 分析应用需求。
- 把基于 MVC 设计模式的 Spring MVC 框架运用到应用中。
- 创建视图组件，包括 hello.jsp（提供 HTML 表单）。
- 创建 messages.properties 资源文件。
- 数据验证，检查用户输入的 userName 是否合法。
- 创建控制器组件：PersonController 类
- 创建模型组件：Person 类
- 创建配置文件：web.xml（Web 应用的配置文件）和 springmvc-servlet.xml（Spring MVC 框架的配置文件）
- 编译、发布和运行 helloapp 应用。

2.1　分析 helloapp 应用的需求

在开发应用时，首先从分析需求入手，列举该应用的各种功能，以及限制条件。helloapp 应用的需求非常简单，包括如下需求：

- 接收用户输入的姓名<userName>，然后返回字符串“Hello <userName>”。
- 数据验证：如果用户没有输入姓名就提交表单，将返回错误信息，提示用户首先输入姓名。
- 模型层负责把表示用户的 Person 对象保存到数据库中。为了简化范例，本范例实际上未实现这一功能。

2.2　运用 Spring MVC 框架

下面把 Spring MVC 框架运用到 helloapp 应用中。Spring MVC 框架可以方便迅速地把一个复杂的应用划分成模型、视图和控制器组件，并且能灵活地操控这些组件的协作流程，简化开发过程。

以下是 helloapp 应用的各个模块的构成。

1．模型

模型包括一个 JavaBean 组件 Person 类，它有一个 userName 属性，代表用户输入的姓名。Person 类提供了 get/set 方法，分别用于读取和设置 userName 属性。Person 类还提供了一个 save()方法，负责把包含 userName 属性的 Person 对象保存到数据库中。

对于更为复杂的 Web 应用，Person 类仅仅包含业务数据，不会访问数据库。访问数据库的操作由专门的 DAO（Data Access Object）数据访问对象来实现，参见本书第 12 章的 12.4 节（创建 DAO 层组件）。

2．视图

视图包括一个 JSP 文件：hello.jsp。hello.jsp 提供用户界面，接收用户输入的姓名，向用户显示“Hello <userName>”信息。如果用户在 hello.jsp 的网页上没有输入姓名就提交表单，将产生错误信息。

3．控制器

控制器包括一个 PersonController 类，它完成三项任务：（1）把数据验证产生的错误消息添加到由 Spring MVC 框架提供的 Model 中，供视图层读取这些数据；（2）调用模型组件 Person 类的 save()方法，保存 Person 对象；（3）把请求转发给视图层的 hello.jsp，由 hello.jsp 生成网页形式的视图。

除了创建模型、视图和控制器组件，还需要创建 Spring MVC 框架的配置文件 springmvc-servlet.xml，它用来设置 Spring MVC 框架的具体工作行为。此外，还需要创建整个 Web 应用的配置文件 web.xml。

2.3 创建视图组件

在本范例中，视图包括一个 JSP 文件：提供用户界面的 hello.jsp。此外，hello.jsp 文件中的所有静态文本存放在专门的 messages.properties 消息资源文件中。

2.3.1 创建 JSP 文件

hello.jsp 提供包含 HTML 表单的用户界面，接收用户输入的姓名。此外，本 Web 应用的所有响应结果也都通过 hello.jsp 来展示给用户。图 2-1 显示了 hello.jsp 生成的网页。

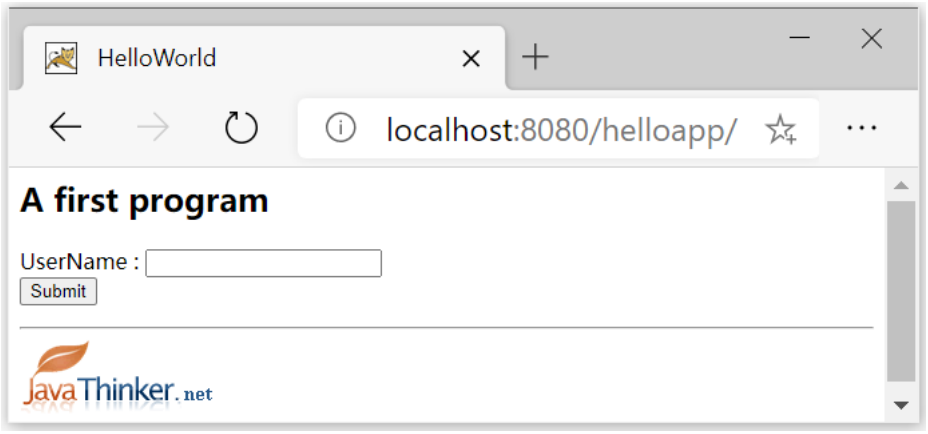

图 2-1 hello.jsp 生成的网页

在图 2-1 中，用户输入姓名“Weiqin”后，提交表单，服务器端将返回“Hello Weiqin”，参见图 2-2。

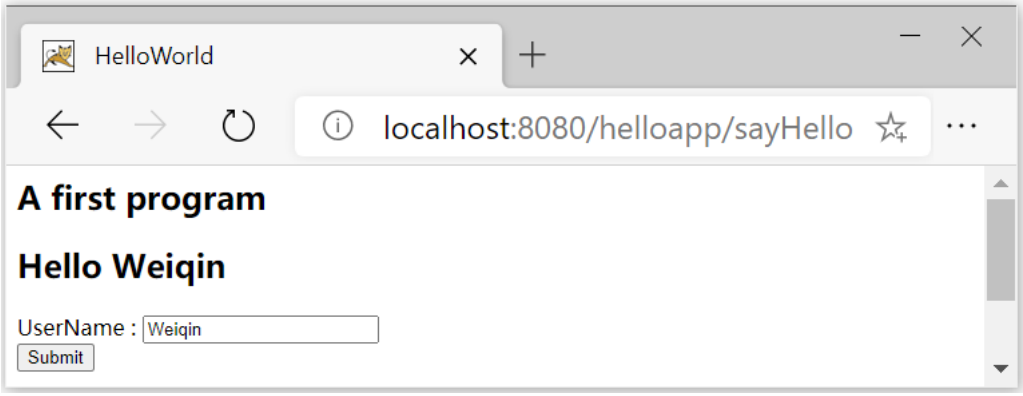

图 2-2　hello.jsp 接收用户输入的姓名后正常返回的网页

例程 2-1 为 hello.jsp 文件的源代码。

例程 2-1　hello.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@taglib uri=
        "http://www.springframework.org/tags/form" prefix="form" %>
<html>
  <head>
    <title><spring:message code="hello.jsp.title"/></title>
  </head>
  <body>

    <h2><spring:message code="hello.jsp.page.heading"/></h2>

    <c:if test="${not empty userName}">
       <h2>
        <spring:message code="hello.jsp.page.hello"/>
        ${userName}
       </h2>
    </c:if>

    <form:form action="${pageContext.request.contextPath}/sayHello"
                modelAttribute="personbean" >

      <spring:message code="hello.jsp.prompt.person"/>
      <form:input path="userName" />
      <font color="red">
        <form:errors path="userName" />
      </font>
      <br>

      <input type="submit" value="Submit"/>

    </form:form>

    <hr>
    <img src=
```

```
            "${pageContext.request.contextPath}/resource/image/logo.gif">
      </body>
    </html>
```

以上基于 Spring MVC 框架的 JSP 文件有以下特点：

- 没有任何 Java 程序代码。
- 使用了 JSP 的 EL 表达式语言来访问各种变量，例如${userName}。
- 使用了来自 JSTL 标签库的自定义标签，例如<c:if>标签。
- 使用了来自 Spring 标签库的自定义标签，例如<spring:message>标签和<form:form>标签等。
- 没有直接提供文本内容，取而代之的是<spring:message>标签，输出到网页上的文本内容都是由<spring:message>标签来生成的。

Spring 标签库的自定义标签是联接视图组件和 Spring MVC 框架中其它组件的纽带。这些标签可以访问或输出由控制器提供的数据。在本书第 4 章（创建 HTML 表单）还会进一步介绍 Spring 自定义标签的用法，本节先简单介绍几种重要的自定义标签。

hello.jsp 开头几行用于声明和加载各种标签库：

```
<%@ page contentType="text/html; charset=UTF-8" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@taglib uri=
        "http://www.springframework.org/tags/form" prefix="form" %>
```

以上代码表明该 JSP 文件使用了 JSTL Core 标签库和 Spring 标签库，这是加载自定义标签库的标准 JSP 语法。

hello.jsp 使用了 Spring 标签库中和 HTML 表单有关的标签，包括<form:form>, <form:input>和<form:errors>：

- <form:errors>：用于输出对表单数据进行验证产生的错误消息。
- <form:form>：用于创建 HTML 表单，它的 modelAttribute 属性取值为“personbean”。<form:form>标签会把 Model 的 personbean 属性包含的数据填充到表单中。这里的 Model 是指 Spring MVC 框架提供的专门用于存放共享数据的模型对象，参见本章 2.4.3 节。
- <form:input>：该标签是<form:form>的子标签，用于创建 HTML 表单的文本框。在本范例中，Model 的 personbean 属性实际上引用一个 Person 对象。<form:input>标签会把这个 Person 对象的 userName 属性赋值给同名的 userName 文本框。

hello.jsp 还使用了 Spring 标签库的<spring:message>标签，用于输出特定的文本内容，它的 code 属性指定消息编号，和消息编号匹配的文本内容来自于专门的消息资源文件，关于消息资源文件的详细用法参见本书第 8 章（Web 应用的国际化）。

hello.jsp 还使用了 JSTL 标签库的<c:if>标签。<c:if>标签用来控制条件判断流程，只有满足条件，才会执行标签主体中的内容：

```
<c:if test="${not empty userName}">
  <h2>
    <spring:message code="hello.jsp.page.hello"/>
    ${userName}
  </h2>
</c:if>
```

在本范例中，<c:if>标签用来判断 userName 变量是否为 null。如果不为 null，就输出

userName 变量。

2.3.2 创建消息资源文件

hello.jsp 使用<spring:message>标签来输出文本内容。这些文本来自于消息资源文件。本例中的消息资源文件为 messages.properties，参见例程 2-2。

例程 2-2 messages.properties 文件

```
hello.jsp.title=HelloWorld
hello.jsp.page.heading=A first program
hello.jsp.prompt.person=UserName :
hello.jsp.page.hello=Hello
person.no.username.error=Please enter a UserName to say hello to!
```

以上文件以“消息编号 code=消息文本”的格式存放数据。对于以下 JSP 代码：

```
<spring:message code="hello.jsp.title"/>
```

<spring:message>标签的 code 属性为“hello.jsp.title”，在 messages.properties 资源文件中与之匹配的内容为：

```
hello.jsp.title= HelloWorld
```

因此，以上<spring:message>标签将把“HelloWorld”输出到网页上。

2.4 创建控制器组件

控制器组件包括 org.springframework.web.servlet.DispatcherServlet 类和自定义的 Controller 控制器类。DispatcherServlet 类是 Spring MVC 框架自带的，它是整个 Spring MVC 框架的中央控制枢纽。Spring MVC 框架允许开发人员创建自定义的 Controller 控制器类，它用来处理特定的 HTTP 请求。

只要把一个 Java 类用 Spring MVC 的@Controller 注解来标识，它就被声明为 Spring MVC 框架的控制器类。如果把 Spring MVC 自带的 DispatcherServlet 比作框架中的总管，那么开发人员自定义的 Controller 就是处理具体流程的干事。这些 Controller 在 Spring MVC 框架中可以方便自如地呼风唤雨，调兵遣将。Controller 既能与视图层进行数据交互，也能与模型层进行数据交互。

例程 2-3 为 PersonController 类的源程序。

例程 2-3 PersonController.java

```
package mypack;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.ui.Model;
import javax.validation.Valid;
import org.springframework.validation.BindingResult;

@Controller
public class PersonController {
```

```
    @RequestMapping(value = {"/input","/"}, method = RequestMethod.GET)
    public String init(Model model) {
        model.addAttribute("personbean",new Person());
        return "hello";
    }

    @RequestMapping(value = "/sayHello", method = RequestMethod.POST)
    public String greet(
            @Valid @ModelAttribute("personbean")Person person,
           BindingResult bindingResult,Model model) {

      if(bindingResult.hasErrors()){
        return "hello";
      }

      //调用 person 对象的 save()方法把 person 对象保存到数据库中
      person.save();

      model.addAttribute("userName", person.getUserName());
      return "hello";
    }
}
```

PersonController.java 是本应用中最复杂的程序，下面分步讲解它的工作机制和流程。

2.4.1 Controller 控制器类的 URL 入口和请求转发

PersonController 类定义了两个方法：init()方法和 greet()方法。在这两个方法前通过 @RequestMapping 注解来设定调用当前方法的 URL 入口：

```
@RequestMapping(value = {"/input","/"}, method = RequestMethod.GET)
public String init() {……}

@RequestMapping(value = "/sayHello", method = RequestMethod.POST)
public String greet(){……}
```

以上代码表明，当浏览器端的请求方式为 GET，并且请求的 URL 为“/input”或者为“/”（表示 Web 应用的根路径）时，Spring MVC 框架会调用 init()方法；当浏览器端的请求方式为 POST，并且请求的 URL 为“/sayHello”，Spring MVC 框架会调用 greet()方法。

明白了调用 PersonController 类的各个方法的 URL 入口，接下来还要了解 PersonController 类如何把请求继续转发给视图层的相关组件。

init()和 greet()方法的返回值都是“hello”，这是特定视图组件的逻辑名字。在本例中，Spring MVC 框架会把它映射为 helloapp/WEB-INF/jsp/hello.jsp。因此执行完 PersonController 的 init()或 greet()方法，Spring MVC 框架会把请求转发给 hello.jsp。

那么，为什么 Spring MVC 框架会很默契地把 init()和 greet()方法的返回值“hello”映射为“hello.jsp”呢？这是因为事先在 Spring MVC 的 springmvc-servlet.xml 配置文件中做了相应的配置：

```
<bean class =
"org.springframework.web.servlet.view.InternalResourceViewResolver">

   <property name = "prefix" value = "/WEB-INF/jsp/" />
```

```
        <property name = "suffix" value = ".jsp" />
    </bean>
```

以上代码配置了一个 InternalResourceViewResolver 视图解析器，这个解析器会把 init()和 greet()方法的返回值“hello”加上前缀“/WEB-INF/jsp/”与后缀“.jsp”。这样，“hello”就映射成为“/WEB-INF/jsp/hello.jsp”。

本范例把 JSP 文件存放在 Web 应用的 WEB-INF/jsp 目录下，这样能提高这些 JSP 文件的安全性。因为浏览器端无法直接访问 Java Web 应用的 WEB-INF 目录下的文件。例如，如果试图通过浏览器访问以下 URL：

```
http://localhost:8080/helloapp/WEB-INF/jsp/hello.jsp
```

服务器端会返回不存在该文件的错误信息。

2.4.2 访问模型组件

Person 对象表示模型数据。PersonController 会调用 Person 对象的 save ()方法向数据库保存当前的 Person 对象：

```
//调用 person 对象的 save()方法把 person 对象保存到数据库中
person.save();
```

在本范例中， PersonController 类仅仅访问了模型组件的简单功能。在实际应用中，Controller 控制器类会访问模型组件，完成更加复杂的功能，例如：

- 通过模型组件到数据库中查询数据，再由视图组件展示这些数据。
- 和多个模型组件交互。
- 依据从模型组件中获得的信息，来决定返回哪个视图组件。

2.4.3 和视图组件共享数据

Controller 控制器类会读取视图层的表单数据，也会把包含响应结果的数据传递给视图层进行展示。Spring MVC 框架用 Model 接口来表示应用程序需要处理或展示的模型数据，模型数据在 Model 中的存放形式为“属性名/属性值”。只要把共享数据存放在 Model 中，视图和控制器就能方便地从 Model 中读取共享数据。

Spring MVC 框架的 Model 接口和 MVC 设计模式中的模型层是不同范畴的概念。Spring MVC 框架的 Model 接口位于控制器层。为了区分 Model 接口和 MVC 设计模式中的模型层，本书中的英文名 Model 特指控制器层的 Model 接口或者它的实例。

如图 2-3 所示，在本范例中，视图层和控制器层共享存放在 Model 中的 Person 对象。

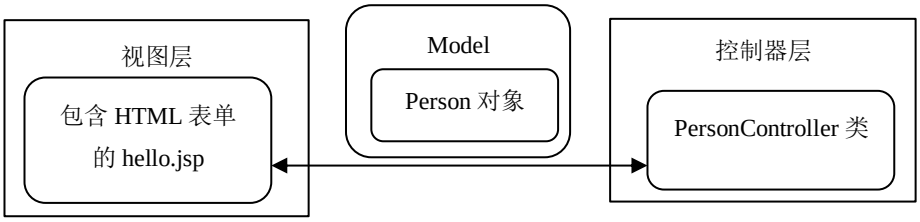

图 2-3 视图层和控制器层共享 Model 中的 Person 对象

Person 对象作为 Model 的一个属性存放在 Model 中，属性名为“personbean”。下面介绍视图层和控制器层如何密切配合，进行你来我往的数据共享和交互。

1．控制器层向视图层传递与表单对应的数据

在 PersonController 类的 init()方法中，向 Model 中存放了一个 Person 对象，属性名为 personbean：

```
public String init(Model model) {
  model.addAttribute("personbean",new Person());
  return "hello";
}
```

在 hello.jsp 文件中，<form:form>标签生成 HTML 表单，它的 modelAttribute 属性用来指定把 Model 的 personbean 属性包含的数据填充到表单中：

```
<form:form action="${pageContext.request.contextPath}/sayHello"
           modelAttribute="personbean" >
```

当 hello.jsp 生成表单时，会读取 Model 中的 personbean 属性，把 personbean 属性包含的数据填充到表单中。personbean 属性引用一个 Person 对象。hello.jsp 把这个 Person 对象的 userName 属性赋值给表单中的 userName 文本框，参见图 2-4。

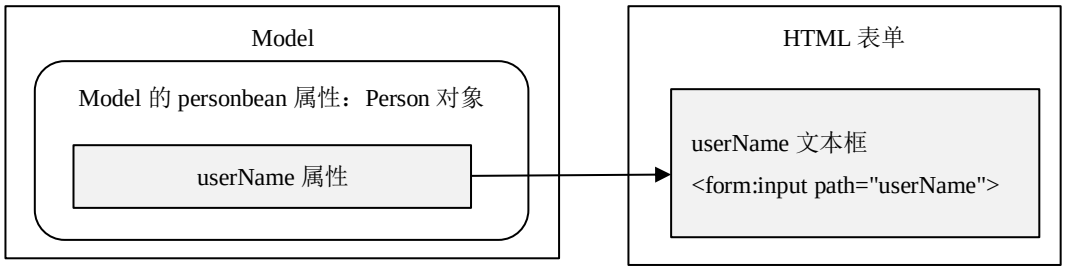

图 2-4　把 Person 对象的 userName 属性赋值给表单的 userName 文本框

以上<form:form>标签的 action 属性指明，当用户在 hello.jsp 的网页上提交表单，该请求由 URL 为“sayHello”的组件来处理，实际上对应 PersonController 类的 greet()方法。

<form:form> 标签的 action 属性值中的${pageContext.request.contextPath}是 JSP 的 EL 表达式，表示当前 Java Web 应用的根路径。

2．控制器层读取视图层的表单数据

在 PersonController 的 greet()方法中定义了一个 person 参数：

```
public String greet(
          @Valid @ModelAttribute("personbean")Person person,
         BindingResult bindingResult,Model model) {……}
```

Spring MVC 框架在调用 greet()方法时，会先自动执行以下操作：

（1）把用户提交的表单数据转换为 Person 对象。表单中 userName 文本框的取值赋值给 Person 对象的 userName 属性。

（2）把 Person 对象赋值给 person 参数。

本书第 3 章的 3.4.6 节（把一组请求参数和一个方法参数绑定）会进一步介绍 Spring MVC 框架把请求参数（包括表单数据）赋值给控制器类的请求处理方法的参数的过程。

在 greet()方法中，就可以从 person 参数中读取表单数据。

3．控制器层向视图层传递各种数据

PersonController 类的 greet()方法的 person 参数使用了@ModelAttribute("personbean")注

解，该注解的作用是把person参数所引用的Person对象存放到Model中，属性名为personbean。

PersonController 类的 greet()方法还有一个 Model 类型的 model 参数，PersonController 类就通过它来向视图层传递数据：

```
//向 Model 中添加 UserName，属性名为“userName”
model.addAttribute("userName", person.getUserName());
```

Spring MVC 框架提供的 Model 接口的实现类在默认情况下，会把这些数据存放在 request 范围内。以上代码的实际作用如下：

```
request.setAttribute("userName", person.getUserName());
```

当 PersonController 把请求转发给 hello.jsp，hello.jsp 就可以方便地获取请求范围内的共享数据：

```
//输出 request 范围内的 userName 变量
${userName}

//把 request 范围内的 personbean 变量的数据填充到表单中
<form:form action="${pageContext.request.contextPath}/sayHello"
                modelAttribute="personbean" >

//把 request 范围内的 personbean 变量的 userName 属性填充到 userName 文本框中
<form:input path="userName" />
```

由此可见，在 Spring MVC 框架的环环相扣的精细布置下，控制器层的控制器类与视图层 JSP 文件中的自定义标签以及 EL 表达式就能够有条不紊地进行数据共享和交互。

确切地说，request 范围的变量是存放在 HttpServletRequest 对象中的一个属性。本书为了叙述的方便，会经常采用 request 范围的变量以及 session 范围的变量等说法，这里的变量和传统的按照 Java 语言声明的变量是有区别的。

2.4.4 Web 组件存取共享数据的原生态方式

在本书中，把视图层组件和控制器层组件统称为 Web 组件。在本章 2.4.3 节，运用 Spring MVC 框架的@ModelAttribute 注解以及 Model 对象，可以实现视图组件和控制器组件之间的数据共享和交互。此外，在 Controller 控制器类中，还可以回归原生态方式，运用 Servlet API 来存取共享数据。

在 Controller 控制器类的请求处理方法中，只要定义一个 HttpServletRequest 类型的 request 参数，就能通过它来存取共享数据：

```
//读取请求参数
String userName=request.getParameter("userName");

//得到 HttpSession 对象
HttpSession session=request.getSession();
//得到 ServletContext 对象
ServletContext context=request.getServletContext();

//存放 request 范围内的共享数据
request.setAttribute("data1","DataInRequest");
//读取 request 范围内的共享数据
String v1=(String)request.getAttribute("data1");
```

```
//存放 session 范围内的共享数据
session.setAttribute("data2","DataInSession");
//读取 session 范围内的共享数据
String v2=(String)session.getAttribute("data2");

//存放 application 范围内的共享数据
context.setAttribute("data3","DataInApplication");
//读取 application 范围内的共享数据
String v3=(String)context.getAttribute("data3");
```

从以上代码看出，从 request 参数可以顺藤摸瓜，得到 HttpSession 对象和 ServletContext 对象，接下来就能随心所欲地在 request 范围、session 范围和 application 范围存取共享数据。

在控制器类的请求处理方法中，除了可以调用 HttpServletRequest 的 getSession()方法来得到 HttpSession 对象，还可以直接在请求处理方法中定义 HttpSession 类型的 session 参数。

以下代码通过 Servlet API 重新实现了 PersonController 类的 greet()方法的部分功能，它只有一个 HttpServletRequest 类型的 request 参数：

```
@RequestMapping(value = "/sayHello", method = RequestMethod.POST)
public String greet(HttpServletRequest request) {
  //读取请求参数
  String userName=request.getParameter("userName");

  Person person=new Person();
  person.setUserName(userName);

  request.setAttribute("userName", userName);
  request.setAttribute("personbean", person);

  return "hello";
}
```

以上 greet()方法通过 request.getParameter()方法读取 hello.jsp 的表单数据，通过 request.setAttribute()方法向 hello.jsp 传递 Person 对象和 userName 对象。但是，以上 greet()方法没有实现对 hello.jsp 的表单数据的验证功能。

Web 组件通过 Servlet API 来存取共享数据，优点是与 Spring 框架松耦合，不依赖于 Spring API，使程序代码具有更好的通用性。缺点是不能与 SpringMVC 框架提供的各种组件紧密协作，例如不能方便地运用 Spring MVC 框架提供的数据验证机制。

Web 组件通过 Spring MVC API 来存取共享数据，优点是使得程序代码不依赖于 Servlet API，而且能与 SpringMVC 框架提供的各种组件紧密协作。缺点是程序代码与 Spring MVC 框架紧耦合，要求开发人员必须对 Spring MVC 本身的运作流程非常熟悉，而且当 Spring MVC API 升级换代时，必须对程序代码进行相应的升级更新。

2.5 创建模型组件

在上一节已经讲过，PersonController 类会访问模型层。在本范例中，模型层包括一个 JavaBean: Person 类。例程 2-4 是 Person 类的源代码：

例程 2-4　Person.java

```
package mypack;
import javax.validation.constraints.*;
import org.hibernate.validator.constraints.NotBlank;

public class Person {
  @NotBlank(message = "{person.no.username.error}")
  private String userName = null;

  public String getUserName() {
    return this.userName;
  }
  public void setUserName(String userName) {
    this.userName = userName;
  }

  /** 把当前 Person 对象保存到数据库  */
  public void save(){  }
}
```

Person 类是一个非常简单的 JavaBean，它包括一个 userName 属性，以及相关的 get/set 方法。此外，它还有一个业务方法 save ()，本例中并没有真正实现这一方法。

通过这个简单的例子，读者可以进一步理解 Spring MVC 框架中使用模型组件的一大优点，它把业务逻辑的实现从 Java Web 应用中单独分离出来，可以提高整个应用的灵活性、可重用性和可扩展性。如果模型组件的实现发生改变，例如本来把 Person 对象的数据保存在 MySQL 数据库中，后来改为保存在 Oracle 数据库中，此时只需要修改模型组件，而不需要对 PersonController 控制器类作任何更改。

Person类还运用了来自Hibernate Validator验证器的@NotBlank注解，用来判断userName属性是否为空：

```
@NotBlank(message = "{person.no.username.error}")
private String userName = null;
```

如果 userName 属性为空，@NotBlank 注解会针对 userName 属性生成一个错误消息，消息文本来自于消息资源文件，以上代码中的{person.no.username.error}指定的是消息 code，在本范例的 messages.properties 资源文件中，对应的消息文本如下：

```
person.no.username.error=Please enter a UserName to say hello to!
```

几乎所有和用户交互的 Web 应用都需要对用户输入的数据进行数据验证，而从头设计并开发完善的数据验证机制往往很费时。幸运的是，Spring MVC 框架提供了现成的、易于使用的数据验证机制，本书第 5 章（数据验证）会进一步详细介绍数据验证的用法。

Spring MVC 框架会统筹安排，调用控制器层和视图层的组件来合作完成数据验证：

（1）在 Spring MVC 框架的配置文件中需要设定真正实现数据验证功能的数据验证器。

（2）在 Controller 控制器类中利用@Valid 注解来声明需要开启数据验证功能。

（3）在视图层的 JSP 文件中通过<form:errors>标签来输出数据验证生成的错误消息。

本章 2.7.2 节还会进一步介绍各个 Web 组件之间紧密配合，进行数据验证的流程。

2.6 创建配置文件

为了让采用了 Spring MVC 框架的 Java Web 应用按部就班地运转起来，还需要提供两个配置文件：

- web.xml：这是 Java Web 应用的配置文件。在这个文件中，通过<servlet>元素隆重邀请 Spring MVC 框架的总管家 DispatcherServlet 粉墨登场，统筹安排 Web 应用的整个运作流程。
- springmvc-servlet.xml：这是 Spring MVC 框架的专有配置文件。Spring MVC 框架的总管家 DispatcherServlet 之所以神通广大，因为它不是孤军奋战，而是依靠许多幕前幕后的得力助手协同工作。在 Spring MVC 的配置文件中需要先为这些得力助手们做一些设置。

Spring MVC 框架以及整个 Spring 框架把这些得力助手称作是 Bean 组件。这里的 Bean 组件不是指传统的 JavaBean，而是指能提供特定服务的 Java 类。由于 Spring 技术发展的早期采用了 JavaBean 的编程风格，后来对各种服务类就一直保持了“Bean”这样的亲切称呼。

2.6.1 创建 Web 应用的配置文件

为了让 Spring MVC 框架掌管 Java Web 应用，需要在 Java Web 应用的配置文件 web.xml 中对 Spring MVC 的中央控制枢纽 DispatcherServlet 类进行配置。例程 2-5 为 web.xml 的源代码。

例程 2-5 web.xml

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
  http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
  version="4.0" >

  <display-name>HelloApp</display-name>

  <servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

</web-app>
```

以上代码中的<load-on-startup>元素确保 Servlet 容器启动 helloapp 应用时，就会初始化 DispatcherServlet。<servlet-mapping>元素的<url-pattern>子元素的取值为“/”，表明所有访问 helloapp 应用的 URL 都会首先由 DispatcherServlet 进行预处理，DispatcherServlet 接下来再把处理客户请求的任务派发给相应的 Controller 组件或其他的 Web 组件。

2.6.2 创建 Spring MVC 框架的配置文件

从本章前面的内容已经看出，Spring MVC 框架全方位地掌管整个 Java Web 应用的请求处理流程、数据验证、视图层和控制器层之间的数据交互，以及消息资源文件等。

Spring MVC 框架之所以功能如此强大，不仅要归功于在幕前大显身手的 DispatcherServer 和 Controller 控制器类，还归功于在幕后默默付出的各种 Bean 组件。Spring MVC 的配置文件会对这些 Bean 组件进行配置。例程 2-6 是本范例中的配置文件 springmvc-servlet.xml 的源代码。

例程 2-6 springmvc-servlet.xml

```
<beans xmlns = ……>
  <!--  指定 Spring MVC 框架扫描 mypack 包以及子包中的 Java 类的注解 -->
  <context:component-scan base-package = "mypack" />

  <bean class = "org.springframework.web.servlet.view
                .InternalResourceViewResolver">
    <property name = "prefix" value = "/WEB-INF/jsp/" />
    <property name = "suffix" value = ".jsp" />
  </bean>

  <bean id="messageSource" class=
           "org.springframework.context.support
           .ResourceBundleMessageSource">
    <property name="basenames">
       <list>
          <value>messages</value>
       </list>
    </property>
  </bean>

   <bean id="validator" class=
           "org.springframework.validation.beanvalidation
           .LocalValidatorFactoryBean">
     <property name="providerClass"
           value="org.hibernate.validator.HibernateValidator" />
     <property name="validationMessageSource" ref="messageSource" />
   </bean>

   <mvc:annotation-driven validator="validator" />
   <mvc:resources location="/"  mapping="/resource/**" />
</beans>
```

以上代码中的<context:component-scan>元素指定 Spring MVC 框架在 Java Web 应用的初始化阶段会扫描 mypack 包以及子包中的 Java 类的注解。Spring MVC 框架需要首先清点 Web 应用中有哪些 Web 组件可以被统一调度。例如 Spring MVC 框架会识别 PersonController 类中的@Controller 注解，从而把 PersonController 类当作可以被调兵遣将的控制器类，纳入自己的管辖中，又根据 PersonController 类中@RequestMapping 注解提供的信息，对访问 PersonController 类的各个方法的 URL 入口一目了然。

以上代码还配置了三个来自于 Spring MVC 框架的 Bean 组件：

- InternalResourceViewResolver：视图解析器，把视图组件的逻辑名字映射为文件系统中实际的视图文件。例如把 PersonController 类的 init()方法的返回值“hello”映射为

“/WEB-INF/jsp/hello.jsp”。

- ResourceBundleMessageSource：消息资源组件。在本例中指定消息资源文件为messages.properties。默认情况下，该文件位于WEB-INF/classes目录下。
- LocalValidatorFactoryBean：数据验证器工厂类。在本例中，它会创建HibernateValidator验证器实例。

Spring MVC配置文件如何起名以及存放路径在哪里呢？这取决于在web.xml文件如何配置DispatcherServlet。Spring MVC配置文件的默认名字为“DispatcherServlet的名字-servlet.xml”。例如，假定在web.xml文件中为DispatcherServlet配置的名字为“springmvc”：

```
<!-- DispatcherServlet配置的名字为“springmvc” -->
<servlet-name>springmvc</servlet-name>
```

那么Spring MVC配置文件的默认名字为“springmvc-servlet.xml”。这个配置文件的默认存放路径为Java Web应用的WEB-INF目录。

在web.xml文件中也可以显式指定Spring MVC配置文件的名字以及存放路径，例如：

```
<servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>classpath:springmvc.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

以上<init-param>元素的<param-value>子元素指定Spring MVC配置文件的名字为springmvc.xml，它的存放路径为Java Web应用的classpath，即WEB-INF/classes目录。

2.6.3 访问静态资源文件

如果在web.xml文件中把DispatcherServlet处理的URL设为“/”，那么在默认情况下，DispatcherServlet会把所有的客户请求都当作是请求访问控制器类的特定方法，如果不存在匹配的方法，就会产生错误。这将导致客户无法正常请求访问静态资源文件，如HTML、TXT、JPG、GIF、JS和CSS等文件。

针对上述问题，Spring MVC框架提供了两种解决方法：

（1）由Servlet容器来处理静态资源文件。

（2）对静态资源文件进行映射，再由Spring MVC框架处理。

1．由Servlet容器来处理静态资源文件

在Spring MVC的配置文件中加入如下元素：

```
<mvc:default-servlet-handler />
```

以上元素会使得Spring MVC框架创建一个org.springframework.web.servlet.resource.DefaultServletHttpRequestHandler对象，它会首先拦截客户请求，如果客户请求访问的是静态资源文件，就把请求交给Servlet容器处理，否则再把请求转发给DispatcherServlet处理。

假定在helloapp应用的根路径的image子目录下有一个logo.gif图片文件。在hello.jsp文件中可以通过以下方式访问logo.gif文件：

```
<img src="${pageContext.request.contextPath}/image/logo.gif">
```

2．对静态资源文件进行映射，再由 Spring MVC 框架处理

在 Spring MVC 的配置文件中加入用于映射静态资源文件的<mvc:resources>元素。例如：

```
<mvc:resources location="/ "  mapping="/resource/**" />
```

以上元素把以“/resource”开头的 URL 和 Web 应用的根路径映射。Spring MVC 框架的 DispatcherServlet 在处理所有的客户请求时，如果检测到 URL 以“/resource”开头，就会到 Web 应用的根路径下读取相应的资源文件。

假定有一个图片文件 logo.gif 的文件路径为：helloapp/image/logo.gif。在 hello.jsp 文件中可以通过以下方式访问 logo.gif 文件：

```
<img src=
   "${pageContext.request.contextPath}/resource/image/logo.gif">
```

2.7　发布和运行 helloapp 应用

helloapp 应用作为 Java Web 应用，它的目录结构应该符合 Java Web 应用的规范，此外，helloapp 应用需要访问以下 JAR 类库文件：

（1）Spring MVC 框架的类库文件，下载地址为：

```
https://repo.spring.io/libs-release-local/org/springframework/spring/
```

（2）JSTL 标签库的类库文件，下载地址为：

```
http://tomcat.apache.org/taglibs/standard
```

（3）Hibernate Validator 验证器的类库文件，下载地址为：

```
http://hibernate.org/validator/
```

图 2-5 显示了 helloapp 应用的目录结构。

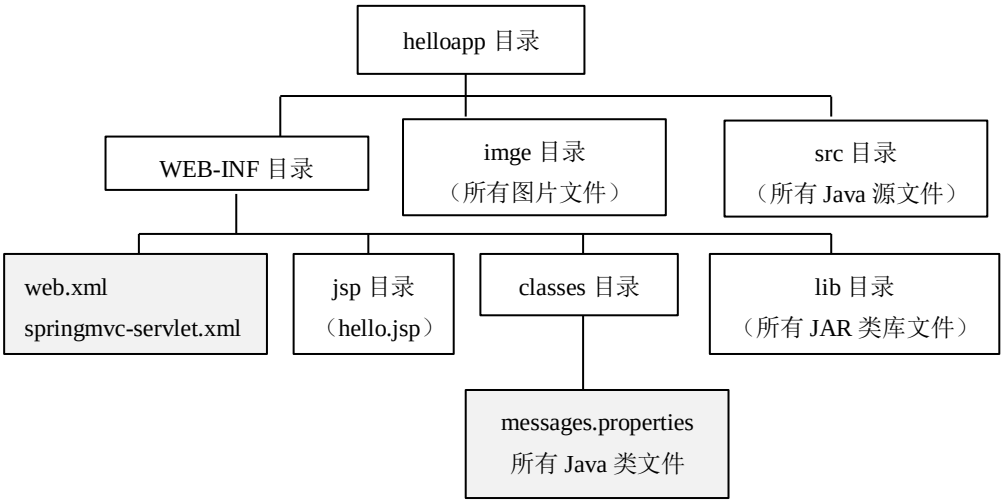

图 2-5　helloapp 应用的目录结构

helloapp 应用的 Java 源文件位于 helloapp/src 目录下，编译这些 Java 源文件时，需要把 WEB-INF/lib 目录中的 JAR 文件加入到 classpath 中。

本书中有些范例的 Java 类还会访问 Servlet API，在这种情况下，还需要把 Servlet API 类库文件加入到用于编译 Java 类的 classpath 中。假定在本地安装了 Tomcat 服务器。在 Tomcat

的根目录的 lib 子目录下提供了 Servlet API 类库文件：servlet-api.jar 文件。

可以用 Eclipse、IntelliJ IDEA、Maven、ANT 等工具来管理、编译和运行本书的范例程序，在本书的技术支持网站（www.javathinker.net）上提供了介绍这些工具用法的参考文档。此外，本书后文还陆陆续续介绍了以下工具的用法：

- 附录 A 的 A.3 节（编译源程序）：用 ANT 工具编译范例。
- 第 13 章的 13.10 节（用 Maven 下载所依赖的类库）：用 Maven 下载应用程序所依赖的类库文件。
- 第 16 章（WebFlux 响应式编程）：用 Itellij IDEA 开发工具开发基于 WebFlux 框架的 Web 应用。
- 第 19 章（用 SpringCloud 发布微服务）：用 Itellij IDEA 开发工具开发基于 Spring Cloud 框架的分布式 Web 应用。

在本书配套源代码包的 sourcecode/chapter02/helloapp 目录下提供了本范例的所有源文件，在 helloapp 应用的根目录下有一个 build.xml 文件，它就是 ANT 的工程管理文件。

只要把整个 helloapp 目录拷贝到 Tomcat 根目录的 webapps 子目录下，就可以按开放式目录结构发布这个应用。

如果 helloapp 应用开发完毕，进入产品发布阶段，应该将整个 Web 应用打包为 WAR 文件，再进行发布。在本例中，也可以按如下步骤在 Tomcat 服务器上发布 helloapp 应用。

（1）在 DOS 下转到 helloapp 应用的根目录。

（2）把整个 Web 应用打包为 helloapp.war 文件，命令如下：

```
jar cvf  helloapp.war  *
```

（3）把 helloapp.war 文件拷贝到 Tomcat 根目录的 webapps 子目录下。

（4）启动 Tomcat 服务器。Tomcat 服务器启动时，会把 webapps 目录下的所有 WAR 文件自动展开为开放式的目录结构。所以 Tomcat 服务器启动后，会发现服务器把 helloapp.war 展开到 Tomcat 的 webapps/helloapp 目录中。

（5）通过浏览器访问 http://localhost:8080/helloapp。

2.7.1 初次访问 hello.jsp 的流程

在 Tomcat 服务器上成功发布了 helloapp 应用后，访问 http://localhost:8080/helloapp 或者 http://localhost:8080/helloapp/input，会看到本章 2.3.1 节的图 2-1 所示的网页。服务器端运行 hello.jsp 网页的流程如下。

（1）DispatcherServlet 调用 PersonController 控制器类的 init()方法。

（2）PersonController 控制器类的 init()方法把一个新建的 Person 对象保存到 Model 类型的 model 参数中：

```
model.addAttribute("personbean",new Person());
```

（3）PersonController 控制器类的 init()方法把请求转发给 hello.jsp。

（4）hello.jsp 中的<spring:message>标签从消息资源文件中读取文本，把它输出到网页上。

（5）hello.jsp 中的<form:form>标签在 Model 中查找属性名为“personbean”的 Person 对象。把 Person 对象中的 userName 属性赋值给 HTML 表单的 userName 文本框。由于此时 Person 对象的 userName 属性的取值为 null，所以在 hello.jsp 网页上，userName 文本框没有内容。

（6）把 hello.jsp 生成的网页视图呈现给浏览器客户。

2.7.2　数据验证的流程

在 hello.jsp 网页上，如果不输入姓名，直接单击【Submit】按钮，会看到如图 2-6 所示的网页。

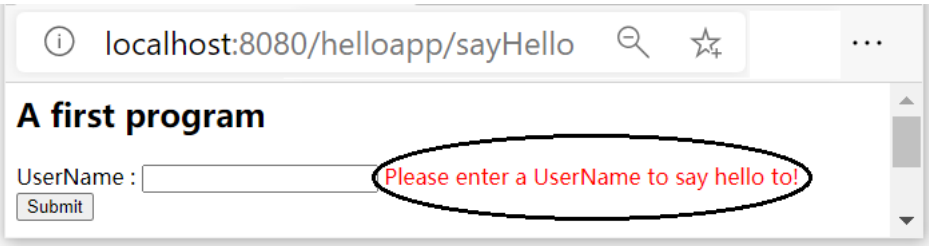

图 2-6　表单数据验证失败的 hello.jsp 网页

当用户提交 hello.jsp 网页上的表单时，请求路径为“/sayHello”：

```
<form:form action="${pageContext.request.contextPath}/sayHello"
modelAttribute="personbean" >
```

服务器端执行数据验证流程如下。

（1）DispatcherServlet 调用 PersonController 控制器类的 greet()方法：

```
public String greet(
        @Valid @ModelAttribute("personbean")Person person,
        BindingResult bindingResult,Model model) {……}
```

（2）由于 greet()方法的 person 参数用@Valid 注解来标识，因此会对 person 对象进行数据验证。

在 Spring MVC 的配置文件 springmvc-servlet.xml 中已经配置了 HibernateValidator 验证器：

```
<!--  配置 id 为“validator”的数据验证器  -->
<bean id="validator"
    class="org.springframework.validation.beanvalidation
            .LocalValidatorFactoryBean">

  <property name="providerClass"
          value="org.hibernate.validator.HibernateValidator" />
  <property name="validationMessageSource" ref="messageSource" />
</bean>

<!--   配置 id 为“messageSource”的消息资源组件 -->
<bean id="messageSource"
   class="org.springframework.context.support
           .ResourceBundleMessageSource">
  <property name="basenames">
    <list>
      <value>messages</value>
    </list>
  </property>
</bean>

<mvc:annotation-driven validator="validator" />
```

以上代码的<mvc:annotation-driven validator="validator" />表明，当 Spring MVC 框架处

理Java类中的注解时，对于具有验证功能的注解（如@NotBlank注解），会采用id为“validator”的验证器来进行数据验证。而id为“validator”的验证器就是上文配置的使用HibernateValidator验证器的Bean组件。

在Person类中，@NotBlank注解用来验证userName属性是否为空：

```
@NotBlank(message = "{person.no.username.error}")
private String userName = null;
```

如果userName属性为空，会产生一个错误消息，Spring MVC框架把这个错误消息存放在一个BindingResult对象中。以上@NotBlank注解的验证功能实际上由HibernateValidator验证器实现。

在springmvc-servlet.xml的配置代码中，id为“validator”的验证器的“validationMessageSource”属性参考id为“messageSource”的消息资源组件：

```
<property name="validationMessageSource" ref="messageSource" />
```

id为“messageSource”的消息资源组件的资源文件为messages.properties。因此以上代码表明数据验证产生的错误消息文本也来自于messages.properties资源文件。对于Person类中的@NotBlank(message = "{person.no.username.error}")，当userName属性为空时，产生的错误消息的编号为“person.no.username.error”，实际上对应的错误消息文本位于messages.properties资源文件中：

```
person.no.username.error=Please enter a UserName to say hello to!
```

（3）如果数据验证失败，PersonController类的greet()方法把请求转发给hello.jsp：

```
if(bindingResult.hasErrors()){
  return "hello";
}
```

hello.jsp通过<form:errors path="userName" />标签输出和userName属性相关的错误消息。

（4）对于步骤（2），如果数据验证成功，PersonController类的greet()方法把Person对象的userName属性值存放在Model类型的model参数中，并把请求转发给hello.jsp：

```
model.addAttribute("userName", person.getUserName());
return "hello";
```

hello.jsp通过EL表达式“${userName}”在网页上输出userName变量。

2.8 依赖注入和控制反转

Spring MVC框架作为Spring框架的分支框架，处处体现了Spring框架所运用的核心开发思想：依赖注入（DI，Dependency Inject）和控制反转（Inversion of Control）。

对于未使用任何第三方框架软件的独立的Java Web应用程序，应用程序本身需要掌管各种对象的生命周期：何时创建对象，何时销毁对象，以及对象与对象之间如何互相关联和依赖。如果运用了Spring MVC框架，应用程序被Spring MVC框架解耦成一个个具有相对独立性的组件，它们不太需要关心其他组件何时创建，何时销毁，身在何方。当一个组件A需要访问另一个组件B时，可以信手捏来。Spring MVC框架会保证组件B随叫随到，随时听候组件A的访问和调用。

如图2-7所示，例如Controller控制器类组件以及Spring MVC配置文件中配置的各种Bean组件，它们的生命周期都是由Spring MVC框架来管理，Web应用程序并没有主动创

建它们，Spring MVC 框架会管理它们的生命周器，并且能得心应手地派遣这些组件来为 Web 应用程序服务。

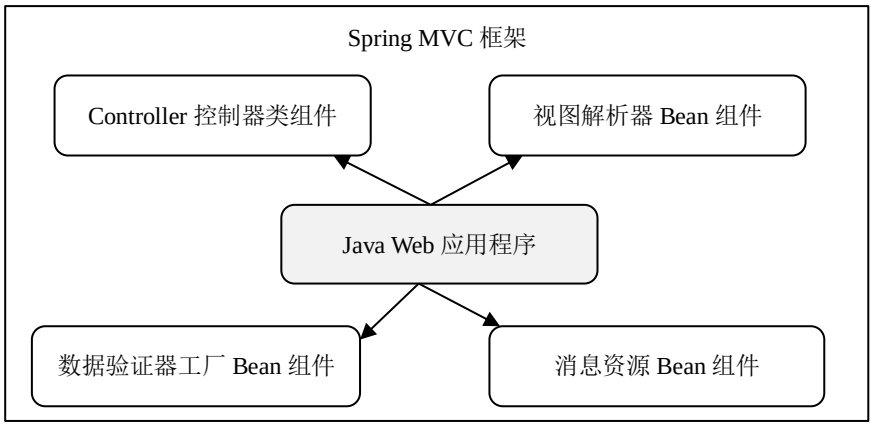

图 2-7　Spring MVC 框架掌管各种组件的生命周期

Spring 框架规定，Bean 组件包括五种作用域：

- singleton：单例作用域。在整个应用程序中，Spring 框架只创建一个 Bean 实例。这是默认的作用域。
- prototype：原型作用域。每次程序访问 Bean 组件时，Spring 框架都会创建一个 Bean 实例。
- request：请求作用域。对于每一个 HTTP 请求，Spring 框架会创建一个 Bean 实例。
- session：会话作用域。对于每一个 HTTP 会话，Spring 框架会创建一个 Bean 实例。
- application：Web 应用作用域。对于整个 Web 应用，Spring 框架会创建一个 Bean 实例。

以上 singleton 和 prototype 作用域适用于所有的应用程序，而 request、session 和 appliaction 作用域仅适用于 Web 应用程序。在 Spring 的配置文件中，<bean>元素的 scope 属性用来设置作用域。例如以下代码配置了一个作用域为 session 的 Bean 组件：

```
<bean id="myCart" class="mypack.ShoppingCart" scope="session" />
```

站在 Java Web 应用程序的角度，Controller 控制器类等组件注入到 Java Web 应用程序中，Java Web 应用程序依赖它们来处理客户请求，却无需管理它们的生命周期，这一过程称作依赖注入。

站在 Spring MVC 框架的角度来看待这一过程，也称作控制反转，也就是说，本来应该由 Java Web 应用程序控制对象生命周期的权力转到了 Spring MVC 框架手中。

有了 Spring MVC 框架从全局角度包揽 Java Web 应用的运作流程，应用程序所创建的组件只需管好各自的一亩三分地，完成本职工作就可以了。

依赖注入和控制反转给软件开发带来以下优点：

（1）软件可维护性好，便于单元测试、调试程序和诊断故障。每一个类都可以单独测试，彼此互不影响。这是组件之间松耦合带来的益处。

（2）软件开发团队中的成员分工明确，职责分明。容易将一个大的任务细分成独立的细小任务，由开发人员分工合作完成，这样能提高软件开发效率。

（3）由于每个组件都具有相对独立性，提高了组件的可重用性。

（4）允许在配置文件中配置 Bean 组件，使得软件能灵活地适应各种需求变化。

2.9　向 Spring 框架注册 Bean 组件的三种方式

为了让 Spring 框架管理特定 Bean 组件的生命周期，首先要向 Spring 框架或它的分支框架注册 Bean 组件。注册 Bean 组件有三种方式：

（1）用@Controller、@Service 和@Respository 等注解标识一个 Java 类。Spring 框架在启动时会识别类中的这些注解，把相应的 Java 类作为 Bean 组件注册到 Spring 框架中。本书第 12 章的 12.6 节（@Repository 注解和@Service 注解）会介绍@Service 和@Respository 注解的用法。

（2）在 Spring 框架或分支框架的配置文件中用<bean>元素来注册 Bean 组件，例如：

```
<bean id="customerService" class="mypack.CustomerServiceImpl" />
```

（3）在程序中用@Bean 注解来注册 Bean 组件，例如：

```
@Configuration
public class MyConfigure {
  @Bean("customerService")
  @Scope("application")
  public CustomerService create(){
    return new CustomerServiceImpl();
  }
}
```

以上 MyConfigure 类用@Configuration 注解来标识，表明它是配置类。create()方法用@Bean("customerService")注解标识，表明该方法注册了一个 id 属性为“customerService”的 Bean 组件，它引用一个 CustomerServiceImpl 对象。@Scope("application")注解表明该 Bean 组件的作用域为 application。本书第 12 章的 16.4 节的例程 16-7（DataHandler.java）也演示了@Bean 注解的用法。

2.10　小结

本章通过简单完整的 helloapp 应用范例，演示了把 Spring MVC 框架运用到 Java Web 应用中的方法。通过这个例子，读者可以掌握以下内容：

- 分析应用需求，把应用分解为模型、视图和控制器来实现这些需求。
- 利用 EL 表达式、JSTL 标签库以及 Spring 标签库来创建视图组件。视图组件中的文本内容保存在专门的消息资源文件中，在 JSP 文件中通过 Spring 标签库的<spring:message>标签来访问它，这样可以很方便地把网页中的文本内容从 JSP 文件中分离出去，提高 JSP 文件的可读性和可维护性。
- 用 Person Bean 把视图中的表单数据传给控制器组件。Person Bean 默认情况下存放在 request 范围内，它能够被 JSP 组件、Spring 标签以及 Controller 控制器类共享。
- 利用@NotBlank 等注解对 Person Bean 进行数据验证。视图组件可以通过<form:errors>标签来访问数据验证产生的错误消息。
- Controller 控制器类调用模型组件来完成业务逻辑，它还能决定把客户请求转发给哪个视图组件。
- 模型组件具有封装业务实现细节的功能，开发者可以方便地修改模型组件的实现方式，这不会对 MVC 的其它模块造成影响。
- Spring MVC 框架提供的 Model 接口帮助 Controller 控制器类把共享数据保存在特

定范围内（默认为 request 范围），从而实现视图组件和控制器组件之间数据的交互与共享。

- 利用 Spring MVC 的配置文件来配置 Spring MVC 框架的各种 Bean 组件。这些在幕后默默付出的 Bean 组件是保证 Spring MVC 框架有条不紊工作的得力助手。

2.11 思考题

1．对于 Person 类中的@NotBlank 注解，以下哪个说法正确？（单选）

（a）它来自于 Spring MVC API

（b）它来自于 Hibernate Validator API

（c）它来自于 Servlet API

（d）它是程序中自定义的注解

2．假定在一个控制器类的请求处理方法中，向 Model 类型的 model 方法参数中存放了一个 Person 对象：

```
model.addAttribute("personbean",person);
```

在 JSP 文件中可以通过哪些方式输出这个 Person 对象的 userName 属性？（多选）

（a）${personbean.userName}

（b）${requestScope.personbean.userName}

（c）<%=((Person)request.getAttribute("personbean")).getUserName() %>

（d）${person.userName}

3．在 Spring MVC 的配置文件中可以配置哪些内容？（多选）

（a）配置消息资源组件

（b）通过<servlet>元素配置 DispatcherServlet

（c）配置数据验证器

（d）配置视图解析器

4．在一个 Controller 控制器类中包含以下两个方法：

```
@RequestMapping(value = {"/hello"}, method = RequestMethod.GET)
public String method1() {……}

@RequestMapping(value = "/hello", method = RequestMethod.POST)
public String method2(){……}
```

以下哪些说法正确？（多选）

（a）当浏览器端请求的 URL 为“/hello”，Spring MVC 框架会依次调用 method1()和 method2()方法。

（b）当浏览器端请求的 URL 为“/hello”，并且请求方式为 GET，Spring MVC 框架会调用 method1()方法。

（c）当浏览器端请求的 URL 为“/hello”，并且请求方式为 POST，Spring MVC 框架会调用 method2()方法。

（d）当浏览器端请求的 URL 为“/method1”，并且请求方式为 GET，Spring MVC 框架会调用 method1()方法。

5．假定在 helloapp 应用的 web.xml 文件中对 DispatcherServlet 做了如下配置：

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <init-param>
     <param-name>contextConfigLocation</param-name>
     <param-value>classpath:springmvc.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

那么 Spring MVC 配置文件的存放路径是什么？（单选）

（a）helloapp/WEB-INF/springmvc.xml

（b）helloapp/WEB-INF/dispatcher-servlet.xml

（c）helloapp/WEB-INF/classes/springmvc-servlet.xml

（d）helloapp/WEB-INF/classes/springmvc.xml

6．对于本章范例中的数据验证流程，在什么情况下会对 Person 对象的 userName 属性进行数据验证？（单选）

（a）程序创建了一个新的 Person 对象。

（b）程序调用了 Person 对象的 setUserName()方法。

（c）Spring MVC 框架开始调用 PersonController 类的 greet()方法时。

（d）用户在 hello.jsp 的网页上输入了表单数据。

7．Spring MVC 框架中的控制器类的默认作用域是什么？（单选）

（a）singleton 作用域

（b）request 作用域

（c）session 作用域

（d）prototype 作用域

参考答案

1．b　2．a,b,c　3．a,c,d　4．b,c　5．d　6．c　7．a